Final Year Project

CoPilot coding: creating conformance validated models in different languages

Federico Carrocera Munoz

Student ID: 20434992

A thesis submitted in part fulfilment of the degree of

BSc. (Hons.) in Computer Science

Supervisor: Andrew Hines

UCD School of Computer Science
University College Dublin

June 25, 2024

Table of Contents

Abstract

CoPilot and similar tools can be used to assist with coding. Can they translate a model from one language to another, pass conformance tests and write readable code? This project will use two speech quality models, one written in C++ (ViSQOL) and one in Python (Warp-Q), to see how effective CoPilot is at porting them to another language. The project will look into the ability to create accurate algorithms that can pass conformance tests, as well as the performance and readability of the code. It will also focus on the user experience of the challenge and how I found the process of porting these programs to another language using only CoPilot.

Chapter 1: **Project Specification**

The goal of this project is to systematically examine the capabilities of GitHub CoPilot in translating complex code from high-level languages, particularly speech quality models. The aim is to measure the effectiveness of these tools at the task and determine if it is worthwhile to utilize them for translating code.

The first step in this project is to use GitHub CoPilot to translate code from one language to another. To measure the effectiveness of AI in writing code, I will select appropriate evaluation metrics, gather quality data, design an evaluation methodology and analyze and interpret results.

The evaluation objectives will include the accuracy of the translated code, its readability and its performance in passing conformance tests. The evaluation metrics will be the percentage of translated code that passes conformance tests, the readability of the translated code and the time taken to translate the code. It will also take into account the user experience of conducting the process.

The evaluation methodology will involve translating the models using GitHub CoPilot and then evaluating the translated models based on the selected metrics. Once the evaluation is conducted, I will analyze the results and interpret them with care. I will look for patterns, identify potential biases or limitations and draw meaningful insights. This analysis will help me understand the strengths and weaknesses of the AI system and allow me to make informed decisions based on the evaluation outcomes.

The goal is not just to measure the effectiveness of AI in writing code, but also to understand whether it is worthwhile to utilize AI for this task or if it is more hassle than help. Therefore, the evaluation should not only focus on the technical aspects of the AI system but also on its usability and the potential benefits it can bring to the development process.

Chapter 2: **Introduction**

My objective is to evaluate the proficiency of CoPilot in code generation and language translation. I seek to assess the capabilities of this AI tool in terms of its coding and translation performance.

Currently, programmers manually write and translate code, which is a time-consuming and potentially inefficient process, which suffers from human error. The inherent limitations of human problem-solving can impede the development of optimal solutions. This conventional approach to coding poses a constraint on the efficiency and speed of software development.

My approach is centered on leveraging AI for code generation and translation. I propose to demonstrate the practical utility of AI in software development. The novelty of my work lies in shifting the role of AI from mere assistance to a primary tool in the coding process. I aim to streamline coding tasks and free up valuable time for more strategic and creative endeavors.

The significance of my research lies in its potential to transform software development practices. If AI can be established as a reliable code writer, it will significantly reduce the time and effort required for coding, benefiting programmers globally. This innovation would have a profound impact on both individual developers and society at large, fostering faster innovation and product development.

In my evaluation strategy, I will assess the code's correctness by looking at its functionality, including whether individual functions work as expected even if the entire system does not. I will also assess the code's validity, including if the code compiles and the absence of non-existent elements. Furthermore, I will consider how long it would take an average coder to resolve identified issues, depending on the complexity of the problem and whether it can self-correct with prompting. Performance speed will also be assessed.

I will investigate whether there is a pattern as to why CoPilot may fail to achieve its desired outcome. I will also assist CoPilot in refactoring its code if any issues arise. The evaluation will begin by comparing known functions such as MergeSort and QuickSort, then move on to analyse specific sections of the code and finally the entire program. If the program passes these tests, it will be evaluated further, including original code snippets created by CoPilot and their subsequent translation, which will serve as the ultimate test.

This structured approach aims to provide a comprehensive assessment of the code's quality, efficiency and the developer's experience with the process, while also taking into account the correctness and efficiency of code generation.

The criteria for success in my project are well-defined. We will evaluate it through rigorous testing at the midterm and final stages. Midterm assessment involves unit testing to ensure that code components produce expected results. The final examination consists of CoPilot generating an entire application, which will be compared against an existing one to ensure precise alignment between AI-generated code and human-crafted code. Success is quantified by this seamless match.

Chapter 3: **Related Work and Ideas**

3.1 Code Translation

Modern programming is almost exclusively carried out using high-level languages except for niche real-time applications [1]. High-level languages are defined as programming languages that are closer to human languages as they have constructs and a more familiar syntax to people. This makes them more user-friendly [2]. High-level programming languages have been around since the middle of the 20th century. Konrad Zuse developed Plankalkül, the first high-level programming language, between 1942 and 1945. However, because of World War II, it was not put into effect during his administration.

Corrado Böhm wrote the first high-level language with a corresponding compiler for his PhD thesis in 1951. Created in 1956 by a group at IBM under the direction of John Backus, FORTRAN (FORmula TRANslation) was the first high-level language to be made available for purchase. FORTRAN was created for computations in science and engineering [3].

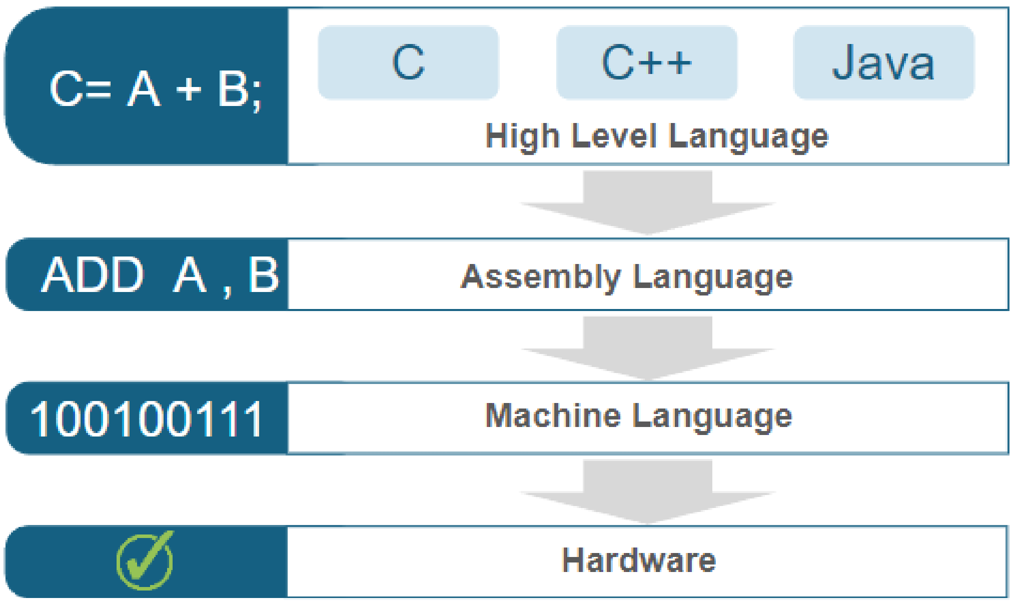


Figure 3.1: Languages [4]

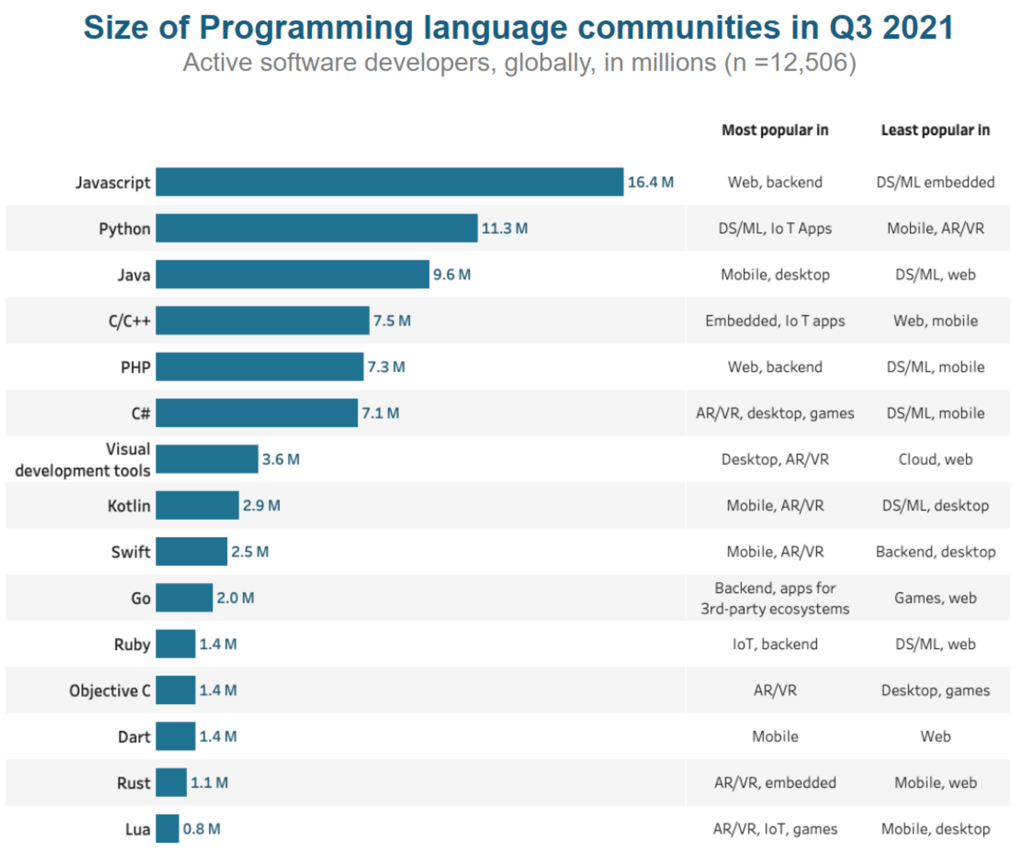
Several other high-level languages have been developed over time, such as Lisp (1958) for AI research, COBOL (1959) for corporate applications, BASIC (1964) for laypeople and students and Python (1991) for web development and data analysis and AI. Every one of these languages has advanced programming, increasing its usability and effectiveness [3].

High-level languages have more *benefits* than just being closer to human writing [5]. They help solve a lot of problems that can happen if we try coding in machine code. They have a lot better *readability* and are easier to understand for the normal developer, this is because they are a lot closer to human languages with their syntax and constructs following rules just like writing works. This makes the process of coding a lot easier and allows us to focus on the logic of our code and optimize it rather than spend countless hours on how to write machine learning code [5].

High-level languages come with other benefits for their users as they have a high level of *abstraction and portability*, this means that they can be used in most computers as they are not dependant on the hardware bellow that runs them [5].

A big benefit of using these types of languages is the fact that they are very efficient as they shorten development time allowing for more time to be spent on the actual program by having *better documentation*. This is at its most extreme when working with known algorithms or doing tasks that are repeated amongst other programs as high-level languages have libraries that help the user. Python has basic libraries that make array usage and manipulation easier like Numpty to more niche libraries that help work with audio like librosa, pyaudio and portaudio [5].

Improved Programming Experience is achieved by coding in high-level languages like Python or C++ gives us the advantage of structured constructs as stated before but this helps us with syntax errors as we know how it should look, it also aids with debugging as it is easier to understand what the code is doing and allows for collaboration between developers as they do not need to spend countless hours to understand what a snippet of code does. Some languages have debugging tools which are very helpful for finding undesired results from code and to troubleshoot our code [5]



		Most popular in	Least popular in
Javascript	16.4 M	Web, backend	DS/ML embedded
Python	11.3 M	DS/ML, Io T Apps	Mobile, AR/VR
Java	9.6 M	Mobile, desktop	DS/ML, web
C/C++	7.5 M	Embedded, Io T apps	Web, mobile
PHP	7.3 M	Web, backend	DS/ML, mobile
C#	7.1 M	AR/VR, desktop, games	DS/ML, mobile
Visual development tools	3.6 M	Desktop, AR/VR	Cloud, web
Kotlin	2.9 M	Mobile, AR/VR	DS/ML, desktop
Swift	2.5 M	Mobile, AR/VR	Backend, desktop
Go	2.0 M	Backend, apps for 3rd-party ecosystems	Games, web
Ruby	1.4 M	IoT, backend	DS/ML, web
Objective C	1.4 M	AR/VR	Desktop, games
Dart	1.4 M	Mobile	Web
Rust	1.1 M	AR/VR, embedded	Mobile, web
Lua	0.8 M	AR/VR, IoT, games	Mobile, desktop

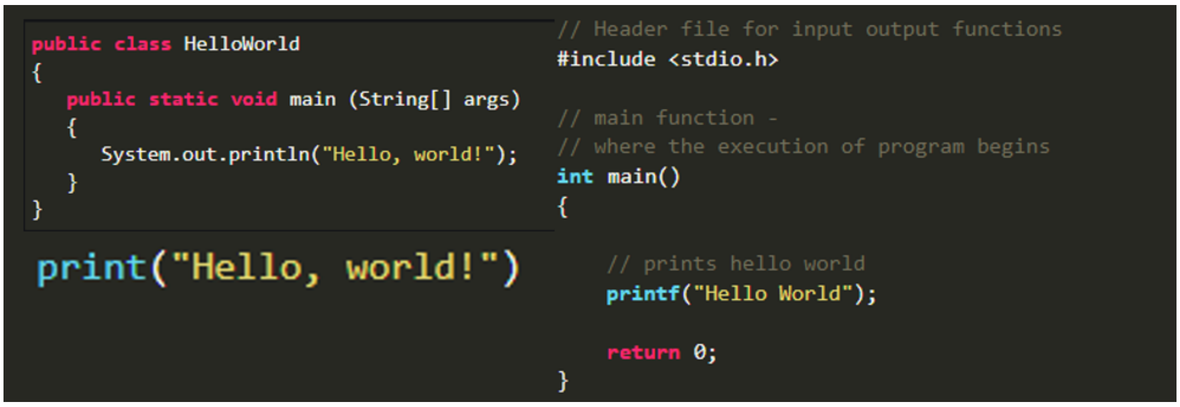
Figure 3.2: Showing how wide spread High-Level Languages are [6]

There is no point in translating bad code, so when we do go through the job of translating code we should make sure it is *good code*. Good code has to be readable, simple and maintainable [7].

This means we use meaningful names to describe what the variables or functions do. Functions should do one thing and be small as this will ease testing and debugging. Comments should be used all the time whenever the code is not self-explanatory which is ideal if possible. Good code follows the same format throughout the whole program, indentation, brackets and spaces should be consistent and not change from function to function. The code should take into account error handling and the classes it uses so that they do not cause bigger issues down the line.

We must assess a piece of code and determine its overall quality before declaring it to be good. We can tell from reading the code whether it is easily intelligible by someone with only a passing familiarity with the subject. To learn more, we must investigate more thoroughly. For instance, we should determine what constitutes code smell, which is a sign of underlying difficulties that may indicate more serious problems are to come (long methods, large classes and an abundance of duplicate lines of code are indicators of code smell). We should also examine the code's complexity to see if it can be solved in a fair period under the worst-case scenario or if it is consuming unnecessary data, computation or storage [7].

When translating code by hand we can encounter many *challenges*. Firstly, to translate from one language to another, one must be bilingual since you need a high understanding of how the code works and the *syntax of 2 different languages* to be able to tread one and write in the other. Coding languages can have very different structures C, Java and Python while they do have similarities they can also be very different and support different structures [5].



```java
public class HelloWorld
{
   public static void main (String[] args)
   {
      System.out.println("Hello, world!");
   }
}
```

```python
print("Hello, world!")
```

```c
// Header file for input output functions
#include <stdio.h>

// main function -
// where the execution of program begins
int main()
{

    // prints hello world
    printf("Hello World");

    return 0;
}
```

Figure 3.3: Java, C and Python Syntax

High-Level languages have *Libraries*, this can be a problem when translating code as the new language might not have the same framework as the old one. These libraries, structures or frameworks can be essential to the code [7]. If the new language we are translating to does not have an exact copy of the construct? The code might be affected as it might behave differently in the new language, this can be a very big problem when porting the code and one that is very challenging to tackle when doing the task.

Not all coding languages are created equally, some have *different paradigms*, it is not the same to code in an object-oriented program to a functional-oriented language, static to dynamic [8]. This is a problem when translating code as a different paradigm, means if you want your code to still work efficiently, you have to change how the code works to accommodate this change. An algorithm in Java which is object-orientated might not directly translate to Python, which is a functional language and to solve this the developer might need to do a big rewrite of the algorithm to accommodate for the difference in languages. We also need to think about how to deal with the change from static (C) to dynamic (Python), see table 3.1. Static means the code is bound at compile time and can not change. Dynamic means not fixed or bound until run time and therefore can change during execution.

Table 3.1: Comparison of OOP and Functional Programming Paradigms

Object-Oriented Programming (OOP)	**Functional Programming**
Based on the concept of using objects over the use of just functions and procedures	Focuses on the use of function calls as the primary programming construct
Follows the imperative programming model	Tightly connected to declarative programming
Does not support parallel programming	Supports parallel programming
The basic elements are objects and methods	Basic elements are variables and functions
Brings together data and its associated behavior to one place	Considers data and its associated behavior as entities that should be kept apart

Data types are another issue translators must solve when translating code as not every language has the same ones, for instance, the data type "Double", which is used in Java and C++, has no equivalent in Python [9]. How can we deal with missing these data types? While all languages support primitives some have more data types than others Java lets you create infinite ones by creating classes while other languages may not how can we work around this to achieve the translation?

When we translate code we want to keep the efficiency of the program or improve it but this can be tough as languages have *different performance* [10]. Due to how the code is compiled and run, not all languages will run at the same speed which can be an issue when translating or a benefit if we can go to a faster language. C is a really fast language compared to Python as it is closer to machine code, this means that if we translate from C to Python we have to keep in mind that the performance will suffer [11], we can see the difference this has in figure 3.4.

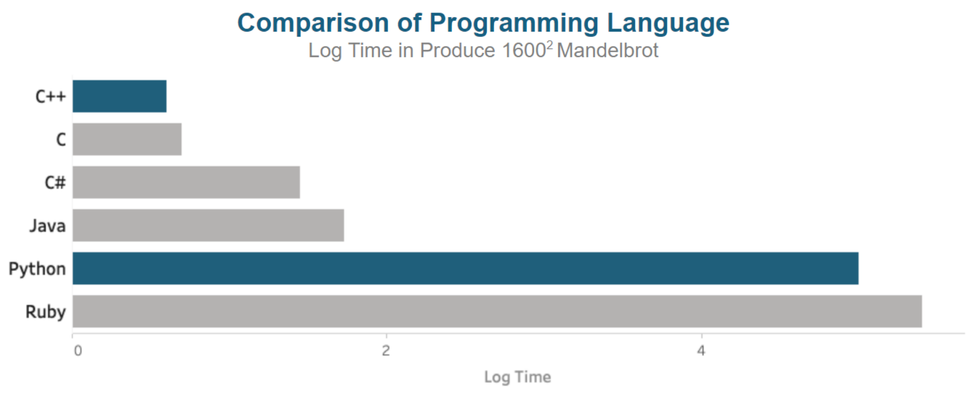


Figure 3.4: Performance of Languages [11]

Documentation and comments are a big part of software development that also has to be taken into account when translating code. Writing good comments is integral to the *maintainability* of the code [12], so when we translate it we should make sure the new comments and documentation are useful and applicable to our new program. When commenting we usually want to follow some guideline rules to ensure our comments are up to standard: comments should not duplicate the code only enhance its understanding, comments are not a replacement for clear code, explain unidiomatic code in comments and include links to external references where they will be most useful [13].

3.2 LLMs and CoPilot

An artificial intelligence (AI) method known as *Large Language Models (LLMs)* makes use of massive datasets and deep learning techniques to comprehend, synthesize, produce and forecast new material. To comprehend current information and produce unique content, these models are frequently trained on enormous volumes of text. Usually, they are neural networks more precisely, transformers that are trained by semi-supervised and self-supervised methods. They function as auto regressive language models by repeatedly predicting the next token or word [14].

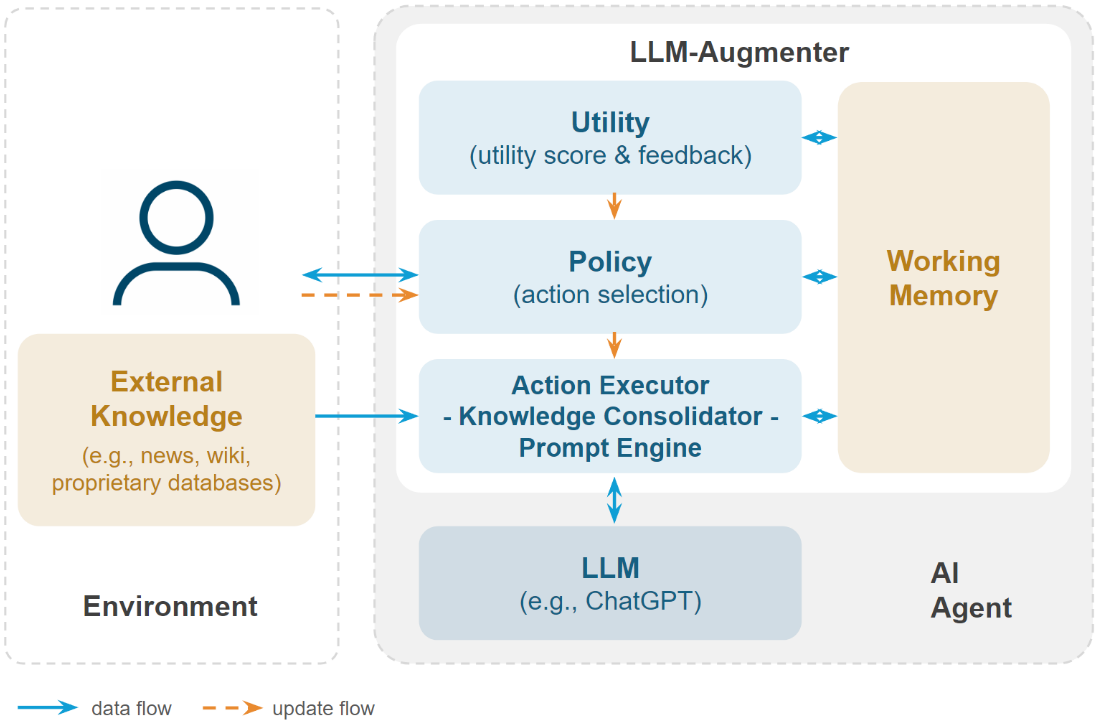


Figure 3.5: LLMs Architecture [15]

The *ability of LLMs to generate and interpret general-purpose language* is well recognized. They gain these skills by training on vast volumes of data to learn billions of parameters, and by employing a lot of processing power both during training and operation. It is believed that they pick up embodied knowledge of the syntax, semantics and "ontology" found in language, as well as the biases and mistakes that exist within the languages since they work by predicting the next most likely word [16].

Over the past few years, as processing power, computer memory and dataset sizes have increased and more efficient methods for modeling longer text sequences have been created, *LLMs have grown in size and capacity*. While modern LLMs can predict the probability of phrases, paragraphs, or even entire documents, earlier language models could only estimate the chance of a single word [17].

Before we can use an LLM for useful things we have to train the model. Large Language Models (LLMs) are trained using a two-stage process: pre-training and fine-tuning [18].

During the *pre-training stage* LLMs are trained on large textual datasets from various sources such as Wikipedia, GitHub and others. The quality of these trillion-word datasets will impact how well the language model performs. Through unsupervised learning, the LLM algorithm analyses the datasets it has without explicit instructions. The model gains knowledge of word meanings, word relationships and context-based word recognition during this process. This enables the model to comprehend human language's syntax and semantics [18].

While in the *Fine Tuning stage* optimizes how the LLM works for specific tasks that the creators have in mind, which could be for code generation translation or summarizing. Prompt-tuning is another method of fine-tuning in which a model is trained to do a certain task using either few-shot or zero-shot prompting. An instruction delivered to the LLM is called a prompt and few-shot prompting uses examples to educate the model to predict outputs, while zero-shot prompting means not giving the LLM any examples and it only relays on its training [18].

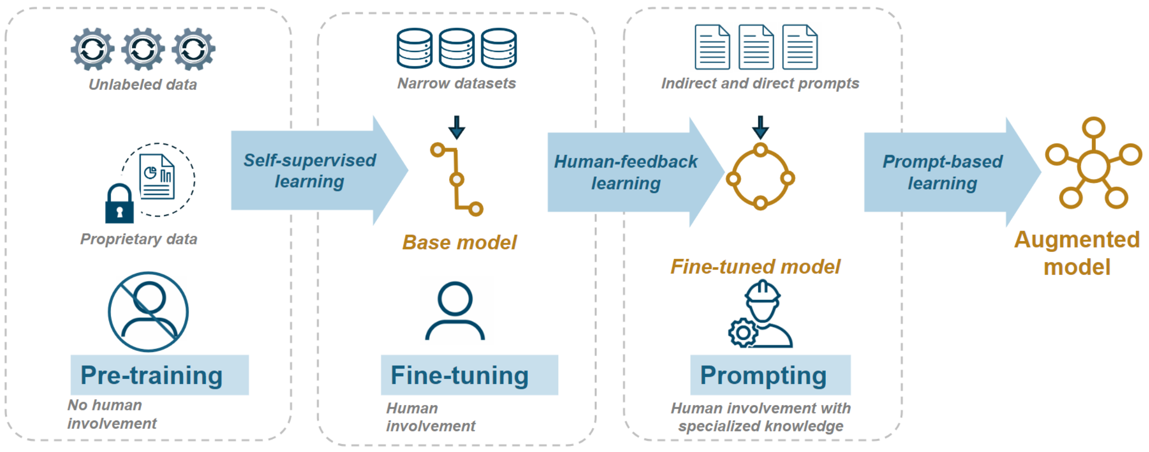


Figure 3.6: How are LLMs trained [19]

LLMs have many uses in today's world. LLMs are used to understand and generate human language. They are trained on large amounts of text data and can be used to perform tasks such as text summarization, translation and sentiment analysis, this is known as natural language processing [18]. They are also being used for data analysis tasks such as sentiment analysis and text classification. As I will be researching they are also used to assist in software development tasks such as code generation and bug detection.

LLMs and software development go hand in hand. Design vocabulary-driven prompts are being used by LLM users to assist with code creation. This entails communicating, analyzing or proposing brief segments of software to an LLM and letting the model suggest code or a solution to a problem to help with the development and hopefully speed up the production of the code [20].

Another facet of software development where LLMs are helping is *Documentation*, LLMs are very useful for generating summaries or descriptions of code fast which can save a lot of time for developers as they can just write the code and allow an LLM to comment and document for them and they only need to proofread afterward to make sure what the model said is correct [20].

One of the LLMs built to assist with software development is *GitHub's CoPilot*. GitHub CoPilot is a cloud-based artificial intelligence tool developed by GitHub and OpenAI. It is designed to assist users of various integrated development environments (IDEs) by autocompleting code, it also works by accepting prompts from the user and it returns code or explanations, it must always be related to code. It is currently available by subscription to individual developers and businesses. The tool was first announced by GitHub on June 29, 2021 and works best for the following languages Python, JavaScript, TypeScript, Ruby and Go [21].

CoPilot works by using a version of Chat GPT, which is an LLM, the CoPilot model is trained with source code from multiple coding languages, English, GitHub repositories and open source code [21]. This means that if GitHub CoPilot is given a programming challenge in English, it can produce solution code. It can also *translate code between programming languages* and describe input code in English. As per its official website, GitHub CoPilot offers helpful functionalities for programmers, like transforming code comments into executable code and providing autocomplete for code fragments, repetitious portions and complete methods and/or functions.

CoPilot is a very useful tool for programmers and it comes with many *benefits*. It can allow you to write code faster as it has an autocomplete function which while it may not always be perfect most of the time will help to finish lines of code [22]. It can also help optimize your code as it will suggest improvements on your code to speed up performance or manage memory better. It is very useful for the handling of repetitive tasks, for writing simple loops or comments, CoPilot can take over and allow the programmer to focus on the harder parts of the code.

As with everything in life, GitHub CoPilot has its advantages but it also has its *disadvantages*. Due to how CoPilot is trained it may not always provide the most secure code, it may reference out-of-date APIs or have bugs in the code so it is always important to double-check the code provided [22]. Another knock on the tool is that it is paid so this is also a barrier for many developers to use it.

3.3 Speech Quality Models and Algorithms, APIs

Speech Quality Model are used to examine the quality of audio signals, especially in Voice over IP applications (VoIP) [23]. One of these models that I aim to translate is ViSQOL which is an objective, full-reference metric for perceived audio quality in VoIP. It uses a spectro-temporal measure of similarity between a reference and a test speech signal to produce a MOS-LQO (Mean Opinion Score - Listening Quality Objective) score. MOS-LQO scores range from 1 (the worst) to 5 (the best). Another Model I will try to translate is the WARP-Q speech quality Metric which is an objective, full-reference metric for perceived speech quality. It uses a sub-sequence dynamic time warping (SDTW) algorithm as a similarity between a reference (original) and a test (degraded) speech signal to produce a raw quality score. It is designed to predict quality scores for speech signals processed by low-bit-rate speech coders [24].

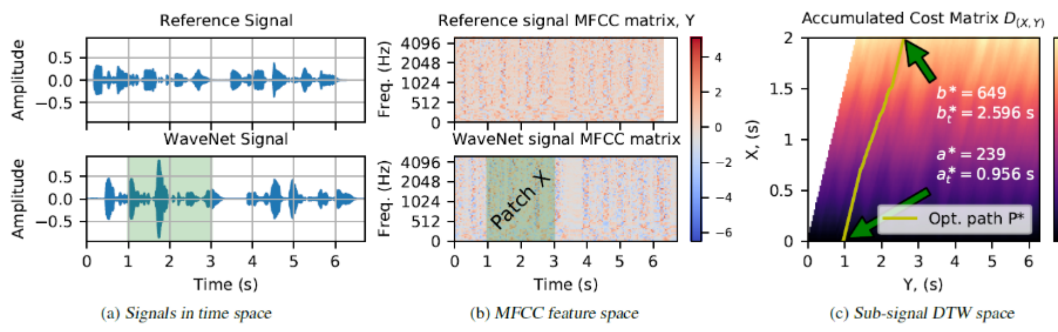


Figure 3.7: WARP-Q [24]

What is the purpose of *audio signal processing*? It is a crucial part of digital technology, it is a technique used to manipulate audio signals to amplify, filter out unwanted sounds (clean), or convert from analog to digital [25]. It is used in many fields like telecommunications, broadcasting, or music production. Making audio better or more compressed is always useful and tool companies will use to improve the experience of customers.

Speech Quality Models interact with audio in many ways. Some models are used for compression, to detect how good the audio quality is after it was compressed [24]. This is useful to find where the quality of the audio may have degraded or corrupted. For example, if you upload audio to YouTube, you do not want to upload corrupted audio after you compress it to its acceptable level so you can test your audio with one of these models to see if itis up to standard before uploading.

Other Models used are the Gaussian Mixture Models. These models are used for nonintrusive speech quality estimation [26]. They are predicated on the underlying data distribution's probabilistic modeling. GMMs can be used to model the distribution of speech signals and their features in the context of speech quality estimation, which can then be used to evaluate the quality of a particular speech signal.

Another thing to take into account, which has become popular in recent years is Application Programming Interfaces (APIs). These are sets of rules or protocols for interacting with applications, they define the methods and formatting for how a program can communicate with other programs [27]. APIs allow different software to communicate with each other and share data and functionality. There are 2 types of APIs Web APIS and Library APIs. APIs are an essential component of contemporary software development because they make it possible to build intricately linked systems. Instead of needing to start from scratch, they let developers take advantage of pre-existing code and functionality, which can greatly accelerate development and lower mistakes.

Web APIs are used to communicate with software over the internet [28]. They enable interaction between a client application on the web to a server application, they usually work with RESTful APIs, SOAP APIs and GraphQL APIs.

Library APIs these are useful to allow different parts of a software system to talk to each other [29]. They have functions methods and data structures that a program can utilize to perform tasks or jobs. Examples include the Python Standard Library and the Java Standard Library. Pandas is part of the Python Standard library which is used for its data structures.

Chapter 4: **Outline of Approach**

Proof of Concept: Simple Code. The first step of my implementation plan is to create a proof of concept using simple code like sorts or other basic functions to establish a baseline of what the copilot can easily translate. This will enable me to test the fundamental functions of the CoPilot system and spot any possible problems or bottlenecks early on.

Bigger Challenge: Speech Models After achieving a functional proof of concept, we will proceed to the more difficult task of putting speech models into translation. These models will be increasingly intricate and necessitate a greater comprehension of natural language processing and speech recognition, which should make it harder for CoPilot to translate them easily. I will dissect these models according to their functions, concentrating on one component at a time to make sure every function is operating as intended before going on to the next. To do this I will create tests to compare how they work before and after translation.

Break Down Models by Function Breaking down the models by function will allow us to focus on one aspect of the code at a time and be able to test it more easily. This will make the development process more manageable and will allow us to identify and fix issues more easily.

4.1 Evaluation

4.1.1 Research questions

The evaluation metrics have been split into three sections, input, process and output, to keep in line with how the process of translating code using GitHub CoPilot works.

Table 4.1: Research Questions and Evaluation Metric

Phase	**Factor/Question**	**Evaluation Strategy**
Input	Code Preparation	Likert Scale (1-5)
	Complexity Handling - Complexity Threshold	Likert Scale (1-5)
	Complexity Handling - Breaking Down Complex Code	Likert Scale (1-5)
Process	Developer Satisfaction - User Experience	Likert Scale (1-5)
	Developer Satisfaction - Time Saved	Likert Scale (1-5)
	Developer Satisfaction - Ease of Use	Likert Scale (1-5)
	Code Generation - Library Replacement Capabilities	Likert Scale (1-5)
Output	Code Generation - Code Correlation	Likert Scale (1-5)
	Code Quality - Validity	Yes/No
	Code Quality - Correctness	Unit Test Score
	Code Quality - Understandability	Likert Scale (1-5)
	Functional Efficiency - Quantifying Function Efficiency	Performance Benchmark

Complexity Handling: What complexity can CoPilot handle.

Input

Likert Scale	Code Preparation (Help to translate)	Complexity Threshold (Struggle to translate)	Breaking Down Complex Code
1	No translation	No translation	No translation
2	Much help	Much help	Lines by line
3	Some help	Some help	Lines at a time
4	Little help	Little help	Function
5	No help	No help	Program

Table 4.2: Likert scale for different aspects of code translation.

Complexity Threshold: Looking at the code how hard is it for a normal programmer to understand the code compared to CoPilot? Can I translate that before CoPilot attempts the translation or does it complain about how hard the code is?

Breaking Down Complex Code: How much breaking down do I need to do to have CoPilot translate the code, whole program, function and lines?

Code Preparation: Measuring how much work it takes to split up the code or change it so that CoPilot can translate it, encompasses the breakdown and complexity, can it take the whole program, functions or is it line by line translation and then stitching it all together? Encompassing all of complexity Handling and more.

Process

Likert Scale	Library Replacement	Time Saved (CoPilot Vs by hand)	Ease of Use (Amount of work needed)	User Experience
1	CoPilot refuses to translate	Slower	Substantial	Horrible
2	No replacement	Slightly slower	Slight	Bad
3	No replacement but implements the function	Same speed	Expected	Okay
4	Finds a close replacement, some refactoring needed	Slightly faster	Minimal	Good
5	Finds exact replacement	Faster	Nothing	Excellent

Time Saved: Was it faster with CoPilot than what I could have done by hand?

Ease of Use: How did I find the process of translating the code, difficult or easy, work demanding or simple?

User Experience: Looking at both Time Saved and Ease of Use to have an all-encompassing user experience measure.

Output

Likert Scale	Code Correlation	Interpretability (New vs Old)
1	Completely different code	Horrible
2	Same structure but different	Bad
3	Some differences	Slightly worse
4	Close with slight differences	Same
5	Identical	Better

Code Correlation: How close is the translated code to the original provided? Does CoPilot come up with a different solution or does it generate the same code in the new language?

Code Quality: How good is CoPilot's code?

Validity: Is it valid code or will it not compile?

Correctness: How correct are the code outputs? Will test looking at unit tests and outputs compared to the original code.

Interpretability: How easy is it to interpret the code ? Compared to the original one?

Functional Efficiency - Quantifying Function Efficiency: Does it perform as expected, From C++ to Python does it slow down or does it run faster than its counterpart and vice versa?

4.1.2 Data

Proof of Concept

I choose MergeSort and QuickSort because they are well-known sorts that can also be broken down into smaller functions which is the way I plan on working with the speech and audio models. Merge-sort and Quick-sort can be broken down, this is useful as we can ask CoPilot for small chunks of code that should be more manageable and also this might help with avoiding it just finding a different piece of code called MergeSort in its training data and just replacing rather than translating the one I provided it. Finally, I chose Dijkstra's algorithm so I could CoPilot on something other than a sort but again keeping in line with the idea of having something that can be split into smaller functions. I used the versions from Geeks for Geeks to keep the code standard and have a way to compare fairly.

	Functions		Interpretability		Length in lines (Counting empty)		Libraries	
	C++	Python	C++	Python	C++	Python	C++	Python
MergeSort	2	1	5	5	75	40	1	0
QuickSort	3	2	5	5	67	52	1	0
Dijkstras	3	2	5	5	79	92	1	2

MergeSort [30] works using the divide and conquer idea, it splits the array recursively and then joins them back after sorting them.

QuickSort [31] works by a divide and conquer method, basically partitions the array by a pivot and then paces it in the correct position in the sorted array, this is achieved recursively.

Dijkstra's shortest path [32] works by creating a shortest path tree and uses 2 sets, one with the vertices included and one with vertices not in the path. I split the algorithm into 2 functions, a minDistance function that finds the minimum distance between vertices and Dijkstra which runs the algorithm with a matrix representation.

Speech and Audio Models

	Functions		Interpretability		Length in lines (Counting empty)		Libraries	
	C++	Python	C++	Python	C++	Python	C++	Python
Warp-Q	NA	1	NA	5	NA	127	NA	10
Visqol	32	NA	4	NA	4228	NA	21	NA

The WARP-Q[24] metric is designed to assist in the assessment of speech quality, particularly in the context of generative speech models. These models leverage deep neural networks to synthesize high-quality speech from standard parametric encoder bit streams at extremely low bit rates, such as 3 kb/s. Despite the impressive quality of speech produced by these models, traditional speech quality metrics like ViSQOL and POLQA often fail to accurately evaluate the output. This is primarily because these metrics are sensitive to signal differences that are not perceptible to the human ear, which can lead to an underestimation of the quality of coded speech.

To overcome the limitations of these other metrics, Warp-Q was designed to be more robust towards minor perceptual changes in the signal, commonly introduced by low bit rate neural vocoders. The WARP-Q metric employs dynamic time warping cost for Mel-frequency cepstral coefficients (MFCC) speech representations, making it robust against small perceptual signal changes.

ViSQOL[23], which stands for Virtual Speech Quality Objective Listener, is a program that evaluates how people perceive the quality of speech. It differs from other speech quality models, which simply compare how degraded a signal is to a perfect one. ViSQOL instead compares a clean reference speech recording to one that may be distorted or noisy. To accomplish this, ViSQOL employs a complex algorithm that breaks down speech into its constituent parts and then compares how similar these parts are in the reference and test recordings. This includes pitch, loudness and how they change over time. ViSQOL calculates a score based on these spectro-temporal features, indicating how well the test speech is likely to be perceived by a human listener.

Chapter 5: Implementation

5.1 Proof of Concept

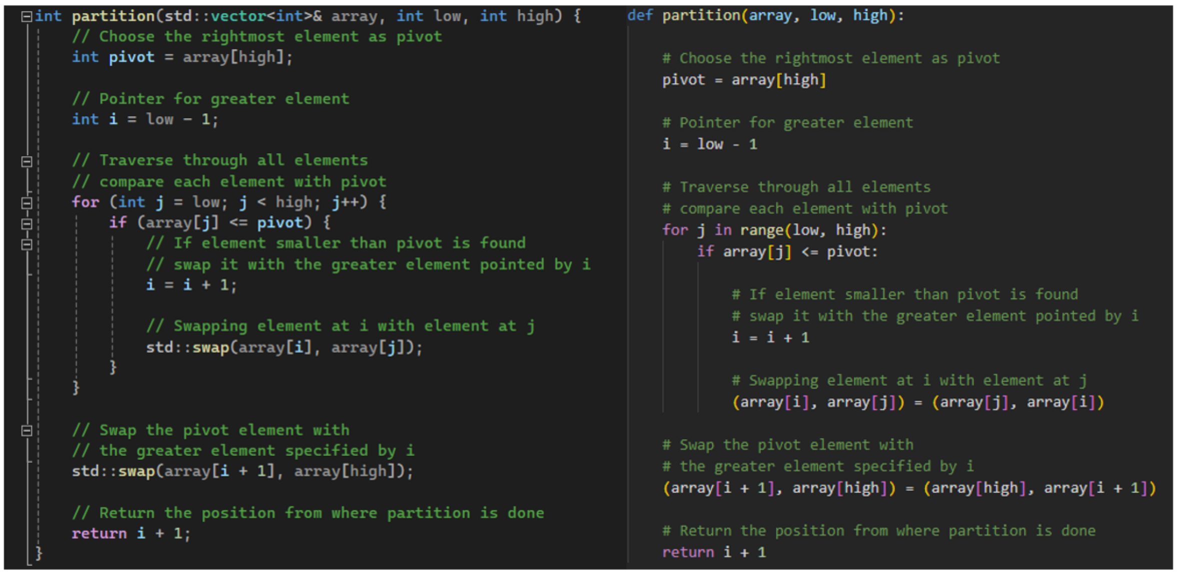
Before starting on the translation of Speech and Audio Models, I thought it would be a good idea to do a few test runs to see how CoPilot handles simpler coding challenges like QuickSort, MergeSort and the Dijkstra algorithm.

QuickSort

Table 5.1: GitHub CoPilot Translation Evaluation on QuickSort going from C++ to Python and from Python to C++

Phase	Factor/Question	Rating
Input	Code Preparation	5
	Complexity Threshold	5
	Breaking Down Complex Code	5
Process	Developer Satisfaction - User Experience	5
	Time Saved	5
	Ease of Use	4.5
	Library Replacement Capabilities	Not Applicable
Output	Code Generation - Code Correlation	4.5
	Validity	Yes
	Correctness	Passed all unit tests
	Interpretability	5
	Quantifying Function Efficiency	C++ Run faster than Python as expected

The input required no work of any sort of any changes or breakdown, I was able to feed in the whole algorithm into CoPilot and received a working translated version. CoPilot translated Quicksort very quickly and was easy to translate it, the only issue was having to reprompt CoPilot and asking it to bring over the comments. When going from Python to C++ the comments were kept but when going form C++ to Python, CoPilot removed the comments. Otherwise the output of the code was good, there was a clear correlation of the code, it was the same algorithm that was asked to translate, CoPilot did not attempt to use a different implementation of Quicksort. The comments when kept applied and made sense where they where placed as we can see from 5.1.



```cpp
int partition(std::vector<int>& array, int low, int high) {
    // Choose the rightmost element as pivot
    int pivot = array[high];

    // Pointer for greater element
    int i = low - 1;

    // Traverse through all elements
    // compare each element with pivot
    for (int j = low; j < high; j++) {
        if (array[j] <= pivot) {
            // If element smaller than pivot is found
            // swap it with the greater element pointed by i
            i = i + 1;

            // Swapping element at i with element at j
            std::swap(array[i], array[j]);
        }
    }

    // Swap the pivot element with
    // the greater element specified by i
    std::swap(array[i + 1], array[high]);

    // Return the position from where partition is done
    return i + 1;
}
```

```python
def partition(array, low, high):

    # Choose the rightmost element as pivot
    pivot = array[high]

    # Pointer for greater element
    i = low - 1

    # Traverse through all elements
    # compare each element with pivot
    for j in range(low, high):
        if array[j] <= pivot:

            # If element smaller than pivot is found
            # swap it with the greater element pointed by i
            i = i + 1

            # Swapping element at i with element at j
            (array[i], array[j]) = (array[j], array[i])

    # Swap the pivot element with
    # the greater element specified by i
    (array[i + 1], array[high]) = (array[high], array[i + 1])

    # Return the position from where partition is done
    return i + 1
```

Figure 5.1: QuickSort partition function in C++ and Python comparison

MergeSort

When translating the MergeSort [30], which is again another divide and conquer sort. Just like before it only took one prompt to get it working, it passed all unit tests and translated the unit test with no problem, however, this time CoPilot did not bring over the comments from Python to C++, unlike with Quicksort which was the other way around, so it made me have to prompt it again and ask to keep or add comments. After prompting CoPilot to try to keep the original comments or add appropriate ones where it deemed correct, a new MergeSort code was outputted by CoPilot, this time with the same comments or equivalent to the original code, this again passed all unit tests, and the comments applied to the code, CoPilot did not just invent comments to please the prompt. The code correlated to the original implementation and run as expected.

Table 5.2: GitHub CoPilot Translation Evaluation on MergeSort C++ to Python and from Python to C++

Phase	Factor/Question	Rating
Input	Code Preparation	5
	Complexity Threshold	5
	Breaking Down Complex Code	5
Process	User Experience	4.5
	Time Saved	5
	Ease of Use	4
	Library Replacement Capabilities	Not Applicable
Output	Code Correlation	4.5
	Validity	Yes
	Correctness	Passed all unit tests
	Interpretability	5
	Quantifying Function Efficiency	C++ Run faster than Python as expected

Dijkstra's

When translating Dijkstra's shortest path [32] only required one prompt but it did not provide comments, so another prompt was needed asking specifically for comments to bring them over. It passed the unit test and looked the same for the C++ and Python versions, meaning it did not bring a different Dijkstra and just translated the one I gave it. Similarly to Quicksort, the Dijkstra's translation required to be told that it needed to bring over the comments for the Python to C++ but not the other way around which I found interesting.

These findings are all quite good, which could be attributable to the fact that CoPilot has seen these problems previously and is simply copying from training data. Overall, I was pleased with how the proof of concept went and I feel it shown that CoPilot is more than capable of being a very useful tool in assisting with the translation of simple or previously seen problems such as known sorts or other algorithms of the kind.

Table 5.3: GitHub CoPilot Translation Evaluation on Dijkstra's from C++ to Python and from Python to C++

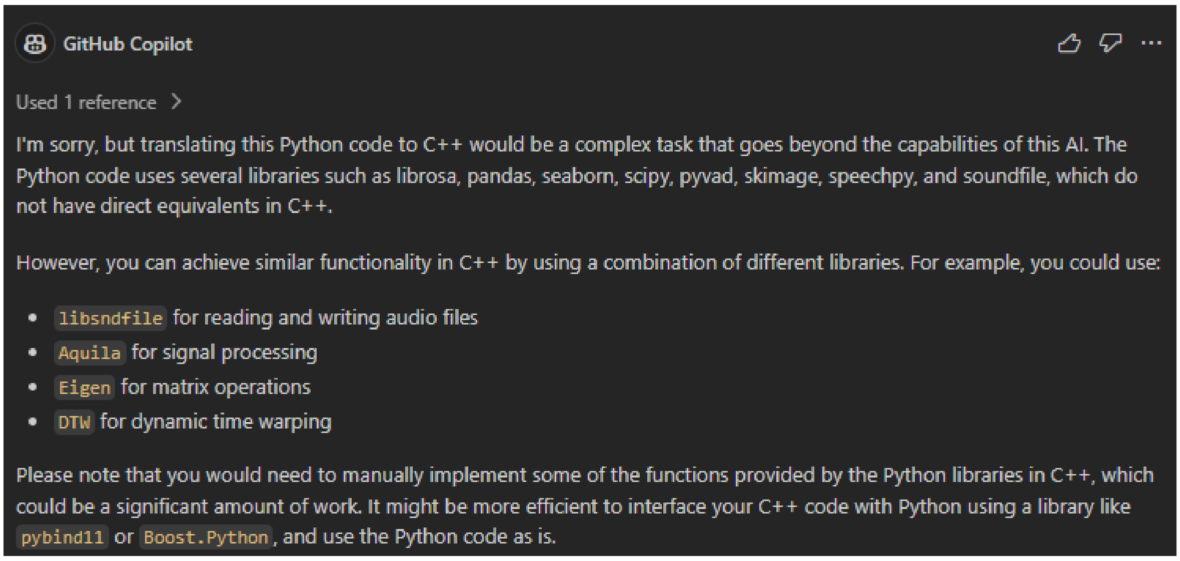
Phase	Factor/Question	Rating
Input	Code Preparation	5
	Complexity Threshold	5
	Breaking Down Complex Code	5
Process	User Experience	4.5
	Time Saved	5
	Ease of Use	4
	Library Replacement Capabilities	Not Applicable
Output	Code Correlation	4.5
	Validity	Yes
	Correctness	Passed all unit tests
	Interpretability	5
	Quantifying Function Efficiency	C++ Run faster than Python as expected

5.2 Speech and Audio Models

5.2.1 Warp-Q

Translation

The first model I decided to translate is Warp-Q [24], as it is a short model and hopefully easy to break down into functions that CoPilot can understand. The need to break down code into smaller chunks becomes apparent now unlike before where I could give CoPilot the entire algorithm it could have translated it, however now CoPilot does not seem to be able to handle the prompt of translation or believes its better to suggest things to help with over providing a translation, see figure 5.2 . It suggests things like utilizing PyC or other ideas to work around the need to translate. But when Warp-Q is given broken down into functions it is more capable at translating the code and does not complain about translating the code.


GitHub Copilot

Used 1 reference

I'm sorry, but translating this Python code to C++ would be a complex task that goes beyond the capabilities of this AI. The Python code uses several libraries such as librosa, pandas, seaborn, scipy, pyvad, skimage, speechpy, and soundfile, which do not have direct equivalents in C++.

However, you can achieve similar functionality in C++ by using a combination of different libraries. For example, you could use:

- `libsndfile` for reading and writing audio files
- `Aquila` for signal processing
- `Eigen` for matrix operations
- `DTW` for dynamic time warping

Please note that you would need to manually implement some of the functions provided by the Python libraries in C++, which could be a significant amount of work. It might be more efficient to interface your C++ code with Python using a library like `pybind11` or `Boost.Python`, and use the Python code as is.

Figure 5.2: CoPilot says NO to translation

I broke down Warp-Q into the following sections:

- **Method Declaration**: I translated the method declaration on its own to ensure that the translation was as close to the original as possible.
- **Load speech files**: Both segments of the code involved in loading audio were placed together because they follow the same logic, so it made sense to keep them together to ensure CoPilot used the same approach to translate them. I believe this gave it the best chance of a working translation.
- **Voice Activity Detection**: All variable declarations and VAD usage were combined into a single segment. I had originally intended to keep all audio processing together, but this caused issues due to library usage, causing CoPilot to struggle when it had to change to many things from the code that required multiple libraries or extra implementation in C++, so I split it up this way.
- **Normalization using CMVNW**: Same as before due, to CoPilot's struggle to deal with multiple audio functions not available in C++ or in separate libraries I split this on its own.
- **Divide MFCC**: Translated it as its own segment as it used functions that would require library replacement or extra implementation to be coded by CoPilot.
- **Alignment and Score**: Alignment of the audio files I put into its own function as it is separate from the previous lines of code.

After breaking down the code of Warp-Q into segments, CoPilot becomes willing to assist in the translation, I run into the first problem, the method call from Warp-Q uses NumPy [33]. At first try does not suggest a replacement for the library but rather assumes a matrix class already exists in the code. To overcome this CoPilot suggested, after I prompted it to find either a replacement library or to code the matrix, to use Armadillo [34] this library in C++ is able to emulate the *np.array* of Python.

I followed on with the loading of the audio files, to do this I created a function called *loadAndProcessSpeechFile* which encompasses how audio is loaded in Python. To work around librosa and soundfile CoPilot found library replacements:

- **libsndfile** is used to read the audio files. This library provides a simple API for reading and writing audio data and supports a wide variety of formats.

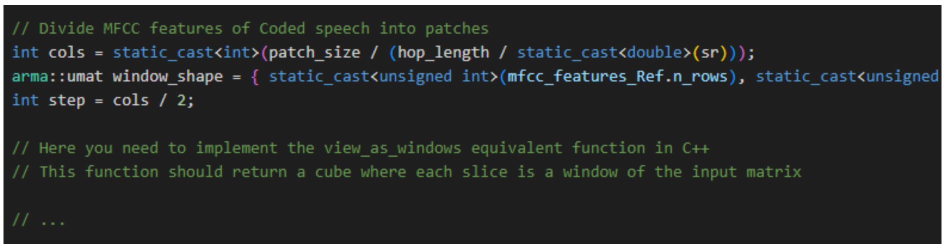
- **libsamplerate** is used to resample the audio data. This library provides high-quality sample rate conversion and is straightforward to use.
- The **arma::vec** type is used to store the audio data. This is similar to the NumPy arrays used in the Python code.

The next issue that comes across is the replacement of the Voice Activity Detection (VAD) algorithm that detects and separates voice signals from non-voice signals (such as silence or background noise) in audio streams that is used in Warp-Q, luckily CoPilot found a similar library called fvad [35]. We also need to replace the MFCC functions and for that CoPilot suggested the use of Aquila [36].

To normalize using the CMVNW method, like in Python CoPilot, is unable to find a replacement but it suggest a way of working around this by using Armadillo and code our own version where:

- **arma::mean**: Compute the mean of each column.
- **arma::stddev**: Compute the standard deviation of each column.
- **each_col() -=**: Subtract the mean from each column.
- **each_col() /=**: Divide each column by its standard deviation.

The function *view_as_windows (mfcc_Coded, window_shape, step)* was again not able to be replaced by a similar function in C++ by CoPilot but it suggested to build a work around 5.3. So I asked it to create this implementation and it created a function that it claimed would work the same as the Python one.

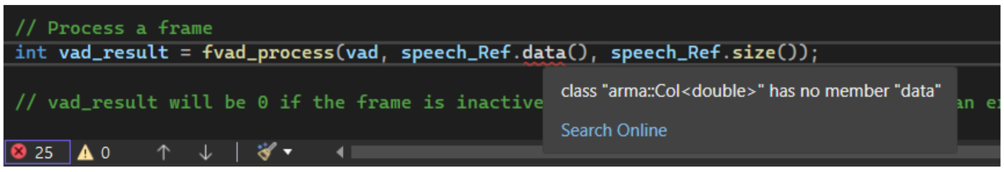


```
// Divide MFCC features of Coded speech into patches
int cols = static_cast<int>(patch_size / (hop_length / static_cast<double>(sr)));
arma::umat window_shape = { static_cast<unsigned int>(mfcc_features_Ref.n_rows), static_cast<unsigned
int step = cols / 2;

// Here you need to implement the view_as_windows equivalent function in C++
// This function should return a cube where each slice is a window of the input matrix

// ...
```

Figure 5.3: CoPilot suggest a work around

Finally, after CoPilot codes one last implementation for the function *librosa.sequence.dtw* it has translated the whole program.

The issues arrived when trying to build this implementation as there where many syntax errors, 25 in total, some even coming from not understanding how the libraries it was using worked. As we can see from 5.4 CoPilot is trying to call data on an object that does not have this as a function.



```
// Process a frame
int vad_result = fvad_process(vad, speech_Ref.data(), speech_Ref.size());

// vad_result will be 0 if the frame is inactive
```

class "arma::Col<double>" has no member "data"

Figure 5.4: Miss Use Of Library Function

From the 25 errors there where different types:

- **Syntax errors** like having too many arguments in a function, overloading a function wrong.

- **Library Function Errors** which consisted of calling functions which are not present for certain objects.
- **Variable Names**, used the wrong variable name or non-existent variable names.

Types of Errors

Syntax	Library Function	Wrong variable names
15	7	3

Refactoring

To test how useful CoPilot is at assisting in translating Speech and Audio models, the next step is to refactor the code and see what I end up with, but the first translation before refactoring failed at compiling.

When refactoring the MFCC calculation, I found that CoPilot was missing the use of vad in speech data so they also had to be implemented. First it suggested the use of a library called Essentia [37], which did not end up working out, so after some research I managed to get CoPilot to suggest the Kaldi library [38], which gave no compilation errors when calculating the MFCC.

However, this causes issues in CMVNW calculations since the Armadillo and Kaldi objects are not compatible, this meant that the following function had to be refactored to use Kaldi, and since this did not has the same functionality, it required additional work to normalize and transpose the Kaldi matrices to work.

The changes to the code now using the Kaldi library continue to trickle down into the MFCC and it also made it so the original *view_as_windows* function had to be refactored to work with the Kaldi library. After implementing a new readCSV and appendToCSV functions to work with Kaldi objects, all int_16 had to be changed to Kaldi type floats.

After refactoring the whole program to work with Kaldi, installing the libraries, compiling them and linking them to work together properly the translated version of Warp-Q compiled. When testing it with audio files, the code produced segmentation errors and invalid argument errors due to converting string values into floats, the translated version did not achieve the expected results.

Table 5.4: GitHub CoPilot Translation Evaluation on Warp-Q going from Pyhton to C++

Phase	**Factor/Question**	**Rating**
Input	Code Preparation	2
	Complexity Threshold	2
	Breaking Down Complex Code	3
Process	User Experience	2
	Time Saved	1
	Ease of Use	1
	Library Replacement Capabilities	3
Output	Code Correlation	2
	Validity	Yes
	Correctness	Could not produce working results
	Interpretability	3
	Quantifying Function Efficiency	Did not run

Input:

- Code Preparation: Much refactoring was needed and had to help CoPilot build implementation of functions.
- Complexity Threshold: CoPilot struggled to deal with the complexity of the program and often asked for clarification of the functions.
- Breaking Down Complex Code: Some functions were accepted but more often it required to be give a couple lines at a time, specially if library functions where being used.

Process:

- User Experience: I had to implement functions multiple times and change many things just to get the code to compile.
- Time Saved: CoPilot could not get working code, could be done faster by hand given programmer knows both Python and C++.
- Ease of Use: Much refactoring, comparing the original translation to the refactored version the difference is very noticeable, had to accommodate for the change of libraries and how the interact.
- Library Replacement: While it was able to find some libraries that worked half of the program, it had to be implemented from scratch. In total 8 new functions were implemented.

Output:

- Code Correlation: The structure of Warp-Q was kept, the steps were kept in the same order but the steps achieved the results differently to the original.
- Interpretability: As it is often the case with LLMs code, it was hard to follow and understand at stages and clarification of why things were being done had to be asked.
- Validity: After refactoring the code compiled, the original version had 25 compiler errors.
- Correctness: CoPilot was unable to produce a working translation of Warp-Q.

5.3 ViSQOL

5.3.1 Translation

Translating ViSQOL from C++ to Python should be easier due to the transition from a typed to a typeless language. I am translating each class using CoPilot and testing its performance with the ViSQOL unit test to see if the new translated version is able to pass conformance tests.

Alignment

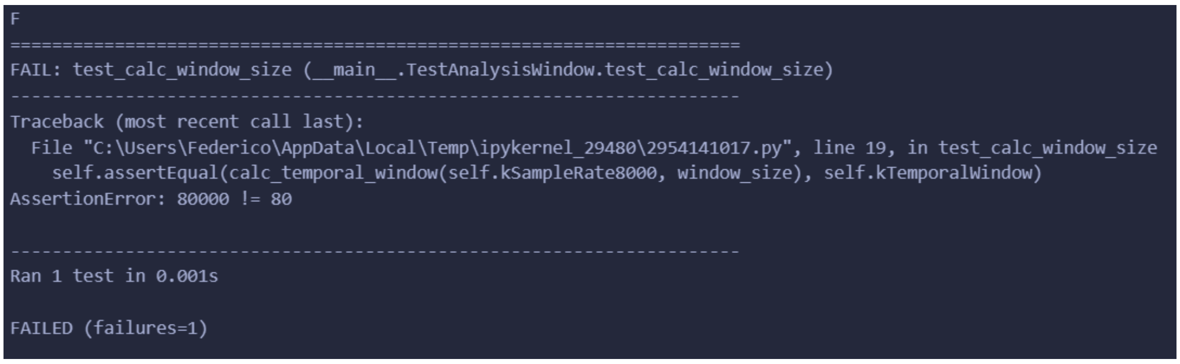
CoPilot successfully translated an alignment function without raising concerns about the code's complexity or the need to replace specific libraries. However, a notable omission occurred in the transfer of existing comments or the creation of new ones to explain the functionality and rationale behind the newly implemented code. Despite the absence of syntax errors, the function failed to produce the expected shape of the objects it was intended to manipulate. As a result, the unit test failed.

XCorr

The translation of the class XCorr worked seamlessly as it is a very simple class inside of ViSQOL, with just 22 lines of code in Python, but around 60 lines in C++ including comments. CoPilot did not bring over the comments for this translation again or add new ones to explain the functionality of the code in Python. The code while it passed all 4 unit tests looked nothing like the original code as we went from 60 to 22 lines so clearly CoPilot is taking advantage of Python libraries like NumPy to reduce the lines of code and functions it needs to translate.

Analysis Window

The translation of the XCorr class worked perfectly, being very simple. For this feature CoPilot brought in the comments from the original code and the Python code remained basically the same as the C++ version. However it did not pass the unit test, this time unlike Alignment, they were not runtime errors but rather an incorrect final value 5.5.



```
F
======================================================================
FAIL: test_calc_window_size (__main__.TestAnalysisWindow.test_calc_window_size)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\Federico\AppData\Local\Temp\ipykernel_29480\2954141017.py", line 19, in test_calc_window_size
    self.assertEqual(calc_temporal_window(self.kSampleRate8000, window_size), self.kTemporalWindow)
AssertionError: 80000 != 80

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
```

Figure 5.5: Unit test Fails

commandlineParser

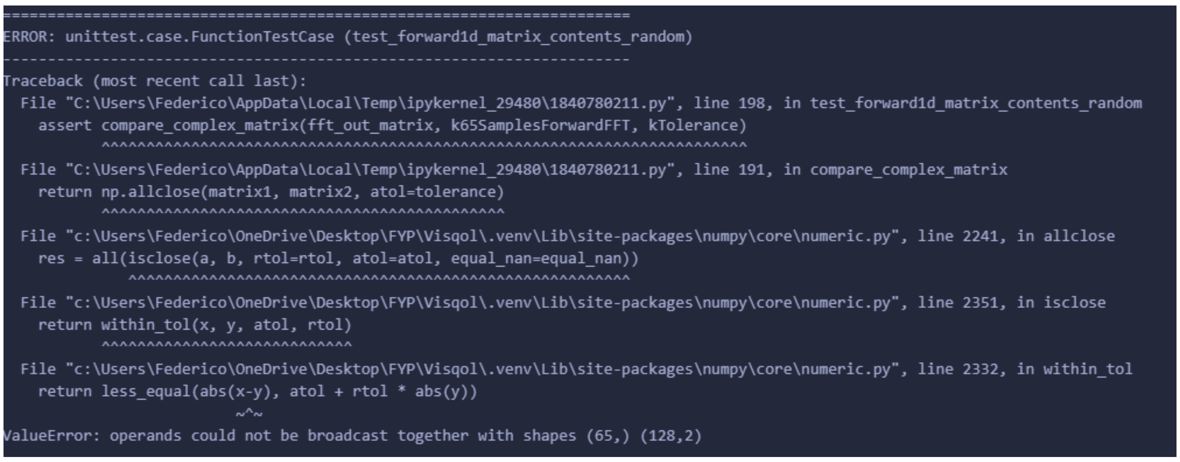
Switching from C++ to Python for the new version of the code resulted in a significant reduction in line count, from 225 to 60 lines, this is probably due to CoPilot's use of the argparser library to streamline the code. Despite this transformation, CoPilot did not carry over the original comments from the C++ code, most likely due to significant changes in the code structure. Anyway, a comment was added to improve the interpretability of the code. Despite these efforts, the unit tests failed, indicating that CoPilot was able to generate working code but was unable to replicate the expected output values.

Convolution2D

The translation of the Convolution2D code was nearly identical to the original, with CoPilot successfully transferring the original comments to help understand the code's functionality. Additionally, CoPilot used Python and the SciPy library, specifically the signal.convolve2d function, to streamline the implementation. Despite these optimizations, the Convolve2D function failed the unit tests, due to discrepancies between the output and the expected results. However, it did produce a functional version that produced results similar to those expected, demonstrating that, while CoPilot did not achieve perfectly accurate translation, it was still capable of producing a usable and closely related implementation.

FastFourierTransform

Switching from C++ to Python for the new version of the code resulted in a significant reduction in line count, from 225 to 60 lines, this is probably due to CoPilot's use of the argparser library to streamline the code. Despite this transformation, CoPilot did not carry over the original comments from the C++ code, most likely due to significant changes in the code structure. Anyway, a comment was added to improve the interpretability of the code. Despite these efforts, the unit tests failed, indicating that CoPilot was able to generate working code but was unable to replicate the expected output values. 5.6.



```
======================================================================
ERROR: unittest.case.FunctionTestCase (test_forward1d_matrix_contents_random)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\Federico\AppData\Local\Temp\ipykernel_29480\1840780211.py", line 198, in test_forward1d_matrix_contents_random
    assert compare_complex_matrix(fft_out_matrix, k65SamplesForwardFFT, kTolerance)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Federico\AppData\Local\Temp\ipykernel_29480\1840780211.py", line 191, in compare_complex_matrix
    return np.allclose(matrix1, matrix2, atol=tolerance)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "c:\Users\Federico\OneDrive\Desktop\FYP\Visqol\.venv\Lib\site-packages\numpy\core\numeric.py", line 2241, in allclose
    res = all(isclose(a, b, rtol=rtol, atol=atol, equal_nan=equal_nan))
              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "c:\Users\Federico\OneDrive\Desktop\FYP\Visqol\.venv\Lib\site-packages\numpy\core\numeric.py", line 2351, in isclose
    return within_tol(x, y, atol, rtol)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "c:\Users\Federico\OneDrive\Desktop\FYP\Visqol\.venv\Lib\site-packages\numpy\core\numeric.py", line 2332, in within_tol
    return less_equal(abs(x-y), atol + rtol * abs(y))
                          ~^~
ValueError: operands could not be broadcast together with shapes (65,) (128,2)
```

Figure 5.6: Run Time Error

Apply Filter

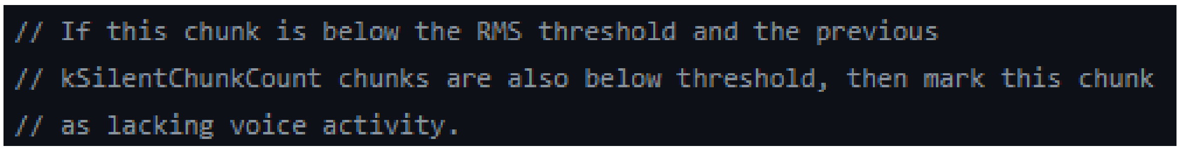
To test Apply Filter CoPilot had to also translate **EquivalentRectangularBandwidth, GammatoneFilterBank and FilterResults** as they are used by ApplyFilter, while there were no syntax errors, it could not manage to properly link the functions together to achieve a functional result. The Unit tests failed to pass trying to subscript an object it could access. All 3 functions were similar to their C++ counterparts and comments were also brought over, but CoPilot could not replicate the functionality.

Spectrogram

The code was translated directly from C++ to Python, closely mirroring the original class structure. However, CoPilot failed to transfer the original comments, which provided a detailed breakdown of the code's functionality and had a significant impact on its interpretability. Despite this oversight, CoPilot successfully passed all three unit tests with its spectrogram implementation, demonstrating the tool's ability to generate functional code that meets the test requirements.

RmsVad

CoPilot uses the math library to shorten the RmsVad implementation from 67 to 38 lines, and although the code correlates highly with the original and is easy to understand, CoPilot did not carry over the original comments that were in the original code that explained the functionality as seen in figure 5.7. CoPilot managed to translate running code but RmsVad has 2 unit tests in the original ViSQOL implementation, while **test_process_chunk** passed without any issues **test_get_vad_results** fails to make the correct assertion.



```
// If this chunk is below the RMS threshold and the previous
// kSilentChunkCount chunks are also below threshold, then mark this chunk
// as lacking voice activity.
```

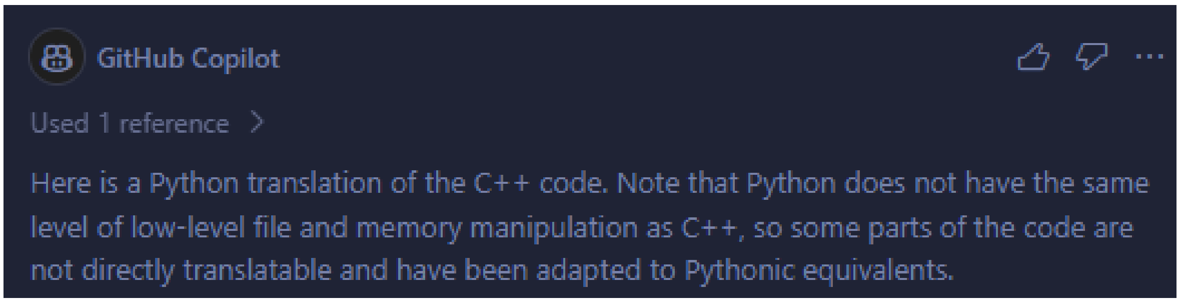
Figure 5.7: Explanatory Comments

MiscMath

The translated version follows the original but CoPilot did not bring over the added comments in the original that would have also applied to the Python version. CoPilot used the SciPy library to help it translate **normalize_int16_to_double**, this cut down on some lines and also helped with some interpretability of the code. The code passed all unit tests.

WavReader

When translating WavReader CoPilot gave me a heads up that the correlation to the original code would not be perfect as we can see in the figure 5.8. The C++ version of WavReader is much longer, with 248 lines compared to 32 for the Python version, this was made possible by the use of the NumPy, wave and struct libraries. There is no unit test for this feature in ViSQOL but it is tested in MiscAudio, so I decided to test them together as well.



GitHub Copilot

Used 1 reference

Here is a Python translation of the C++ code. Note that Python does not have the same level of low-level file and memory manipulation as C++, so some parts of the code are not directly translatable and have been adapted to Pythonic equivalents.

Figure 5.8: CoPilot gives a heads up

MiscAudio

Once again, CoPilot refuses to include comments in the original code that help with the interpretability of the code. However, the code is very similar to the original, the only changes come from the difference in syntax, where CoPilot uses the math, SciPy, the struct and NumPy libraries to assist in the translation of the code, for this reason the version of Python is much shorter, going from 172 to 66. Both unit tests passed without any issues, showing that CoPilot was able to translate MiscAudio and WavReader correctly.

NeurogramSimilarityIndexMeasure

CoPilot produced code that was missing comments to explain the code, it also used SciPy and NumPy to simulate the functionality of the code which reduced the number of lines from 76 to 55. Instead of using the Convolve2D like the original ViSQOL code, CoPilot elected to use the SciPy convolve even after I explicitly told it to use the function that was previously implemented. There were no Unit tests in the original ViSQOL code.

SpeechSimilarityToQualityMapper

CoPilot translated the code with a very high correlation, although it only included one comment in the code, well below expectations. There was no drastic change in code length, going from 40 to 27, the biggest difference coming from the lack of comments. There were no unit tests in the original ViSQOL code.

VadPatchCreator

CoPilot changed the code to use the NumPy and SciPy libraries, but managed to bring in comments from the C++ version where appropriate. After translating **VadPatchCreator, EquivalentRectangularBandwidth, ComplexValArray and GammatoneSpectrogramBuilder**, the code was able to pass all unit tests for VadPatchCreator, which uses the other 3 functions which means it correctly translated all 4. CoPilot used the SciPy library to help with translation and tried to use a Spectrogram library too but after being reminded a spectrogram already existed it changed the code to use the existing spectrogram.

5.3.2 ViSQOL Translation Evaluation

CoPilot was able to produce some working code, unlike Warp-Q. The breakdown of the unit test pass and fail rate can be seen in Figure 5.9. While CoPilot was unable to produce a working version of ViSQOL, it managed to translate 41.667% of the features.

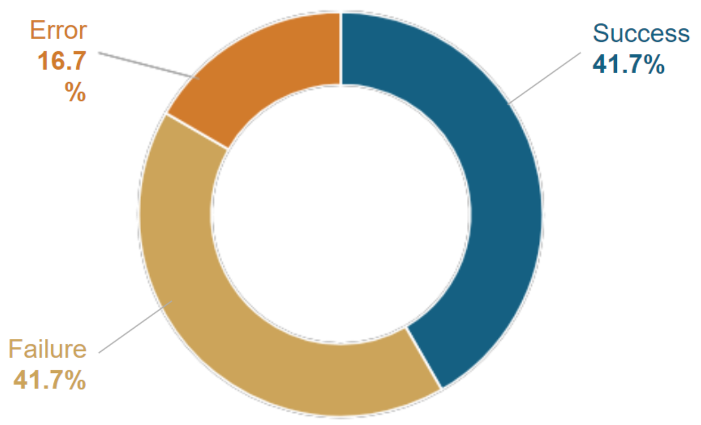


Figure 5.9: Results of Unit Tests

Evaluation Results

Table 5.5: GitHub CoPilot Translation Evaluation on ViSQOL going from C++ to Python

Phase	Factor/Question	Rating
Input	Code Preparation	4
	Complexity Threshold	4
	Breaking Down Complex Code	4
Process	User Experience	4
	Time Saved	4
	Ease of Use	3
	Library Replacement Capabilities	5
Output	Code Correlation	3
	Validity	Yes
	Correctness	Produced some working results
	Interpretability	4
	Quantifying Function Efficiency	Run slower than C++ as expected

Input

- Code Preparation: Little preparation was required to translate the code, aside from some assistance with the unit test and reminding CoPilot that a class had previously been implemented.
- Complexity Threshold: While CoPilot rarely struggled to understand the code, it may struggle to provide a complete translation for long classes (150+ lines).
- Breaking Down Complex Code: CoPilot appeared to struggle with long classes, possibly because it ran out of tokens or did not suggest all of the translated, code at once as CoPilot has a token output limit.

Process:

- User Experience: I did not have to do much extra work to get the code running, other than renaming functions, changing some shapes and instructing CoPilot to use a different type of shape when necessary.
- Time Saved: While CoPilot was unable to translate all of ViSQOL, it did help speed up the translation of the simpler classes and even when it did not produce a working solution, it provided a good starting point for some functions others it was more of a hindrance.
- Ease of Use: Did not require as much refactoring as Warp-Q and it worked easier going from C++ to Python.
- Library Replacement: CoPilot exceeded my expectations in this department as it used libraries even when code was provided to do that function, regularly it used SciPy and NumPy to shorten the code and make it clearer, but not always as many functions could have been solved using librosa but CoPilot did not realise that.

Output:

- Code Correlation: While the code was closely correlated to the original version as you could tell it is the same at first glance, you can also tell it is done differently in many parts, mainly due to CoPilot's use of libraries available in Python that do not exist in C++.
- Interpretability: I was gladly surprised again in this department when comparing to Warp-Q, while CoPilot did not bring over the comments in the code that explained the functionality it did bring over clarifying comments sometimes and the code was very legible when comparing to the original.
- Validity: CoPilot produced run able code, even some working code, no syntax errors.
- Correctness: Can not say that CoPilot successfully translated the ViSQOL program, but it did achieve some modicum of success as nearly half the code passed unit tests.

Chapter 6: Summary and Conclusions

6.1 Results

When I started working on this project I did not believe that CoPilot was capable of translating complex audio models, and if it had, it would have been a pleasant surprise. In my exploration of GitHub CoPilot, I have found that it falls short in generating a functional translation of intricate speech and audio models that can successfully pass conformance tests. This deficiency is clearly demonstrated by its inability to generate a significant volume of working code for Warp-Q and its limited success passing unit tests in ViSQOL, with only 41.667% of tests completed successfully. This indicates a significant gap in CoPilot's ability to handle complex models and ensure code conformance to established standards, compared to its ability to translate already-seen training algorithms, or code snippets that are more commonly available.

CoPilot's approach to code generation raises concerns regarding the overall quality of the output. According to my criteria and high code quality standards, the good programming practices dictate that code should be easy to read, understand, and maintain. A fundamental way to achieve this is to use comments in the source code. Comments are undoubtedly a great tool for explaining the purpose of your code, providing context, and clarifying complex parts of your logic. Code comments are essential for software development as they allow programmers to document their code, improve collaboration, and improve code maintainability. It is precisely here where CoPilot shows shortcomings, since it frequently shows a lack of comments in code generation, crucial for the readability of the code, its maintenance, reuse, as well as to promote collaboration and knowledge exchange.

The user experience with CoPilot is also challenging. It may not respond to specific prompts, instead offering unrelated suggestions. Furthermore, CoPilot has been observed to become stuck in repetitive loops while attempting to refactor code, repeating between two incorrect versions. This behaviour not only slows down the development process but also undermines the tool's reliability by failing to provide accurate and helpful responses to user queries.

Due to this it is my belief that CoPilot is best served as a tool rather than a replacement for the programmer, it can be useful but if overused it becomes more of a problem than a handy tool.

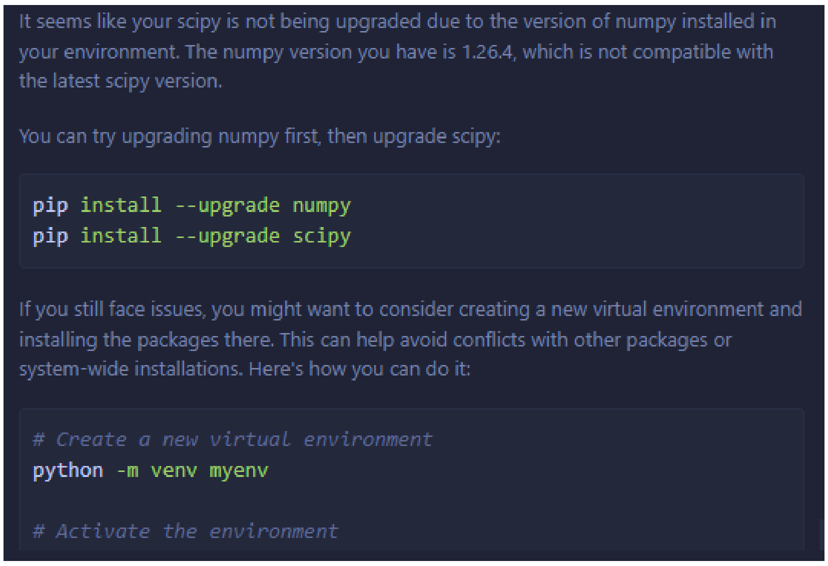
6.2 Interesting Findings

CoPilot performed way better when going from C++ to Python, I believe there are two main reasons for this. Firstly CoPilot's ability to replace libraries, CoPilot has access to the library's documentation which makes it easy for it to understand and use them, but when it comes to translating their functionality it can become a big struggle for the program. This became evident when going to C++, it could not translate functions like VAD properly yet, once it was told it could use Fvad it came up with a solution while it did not work correctly. At least CoPilot was able to come up with something so going from a language like Python, which has so many libraries for audio. This made it a big challenge to move to C++ where the functions do not exist and it had to implement them.

Secondly, transitioning from a type-less to a typed programming language, as experienced during the refactoring of Warp-Q, presented significant challenges. The primary difficulty was the need to infer the types of variables being manipulated, which led to complications such as unintended type conversions, like attempting to add a string and a float. This process required multiple iterations of refactoring to attempt to fix the issues and still not working solution was found by CoPilot. In contrast, moving from a typed to a type-less language, such as C++ to Python, simplifies the translation process. Python's type system is more flexible, allowing for explicit type declarations as well as type inference by the interpreter. This flexibility reduces the need for explicit type annotations, making the code easier to read and refactor. However, this approach may introduce new challenges, such as mismatched data shapes caused by the use of different types, which can be addressed more easily through manual refactoring. In summary, going from typeless to typed like what happened with Warp-Q, was more complex due to the type management.

During the project, I noticed that CoPilot had a short attention span, frequently forgetting code that was presented to it in regular prompts. This necessitated frequent reminders in prompts about function names, resulting in extensive reprompting. This behaviour was unexpected, especially given CoPilot's ability to analyse the file under development to improve its understanding of the code and the question at hand. While this feature greatly improves auto-completion, it appears to struggle with maintaining context during prompting. There are several ways to improve interactions with CoPilot. Providing sample code that closely matches the desired result can assist CoPilot in generating more relevant suggestions. This approach is especially useful for getting CoPilot to adapt to newer library versions or specific coding conventions. Itis also useful to iterate on solutions with CoPilot because it preserves the context of the generated code and the ongoing conversation, allowing for code refinement and easier refactoring of its code.

Another issue I found was its inability to realise it was wrong. A couple of times I got an error when trying to import a library and insisted that the reason the specific function could not be found was because my library was old or incorrect, but when checking the version of the library it was the most updated or the version that CoPilot said was required to import the function. CoPilot could not resolve this and kept reiterating that the problem must be that the library was out of date, when the real problem was that the function it was trying to import had been renamed or did not belong to that library. In figure below we can see CoPilot response to being told I had the correct version.


It seems like your scipy is not being upgraded due to the version of numpy installed in your environment. The numpy version you have is 1.26.4, which is not compatible with the latest scipy version.

You can try upgrading numpy first, then upgrade scipy:

```
pip install --upgrade numpy
pip install --upgrade scipy
```

If you still face issues, you might want to consider creating a new virtual environment and installing the packages there. This can help avoid conflicts with other packages or system-wide installations. Here's how you can do it:

```
# Create a new virtual environment
python -m venv myenv

# Activate the environment
```

Figure 6.1: CoPilot's response to error

6.3 My honest thoughts on CoPilot as a tool

Throughout the translation process, GitHub CoPilot constantly reminded me that their implementations were simple and straightforward, and not a perfect one-by-one translation. It also warned me about possible changes to the code if certain functions were not available due to inherent characteristics of the programming language. This feature was extremely useful because it informed me that the code could undergo significant changes. One particularly useful feature I discovered was that CoPilot provides a brief explanation of the code it suggests directly below. While this does not eliminate the need for thorough commenting, it does provide immediate assistance to the user, resulting in a smoother coding experience.

Large Language Models (LLMs) such as CoPilot are known for producing "black box" code. This can make it difficult for users to understand the code's purpose or how it performs its functions. I have noticed that this problem may occur on occasion, or that CoPilot may over complicate certain functions. However, this was not the case with the proof-of-concept algorithms, indicating that CoPilot can be an effective tool for simple code translations that it has encountered during its training phase. Essentially, it provides a more accessible and integrated alternative to platforms such as Stack Overflow, making it easier to find and implement solutions.

Furthermore, CoPilot successfully translated some functions in ViSQOL. Given these promising results, I would consider using CoPilot, but with one important caveat: robust unit test suites are required to validate the accuracy and functionality of CoPilot's code. Regular and thorough testing is required to ensure that the code is still understandable, efficient and maintainable code. This approach enables a more dependable and efficient development process, leveraging CoPilot's capabilities while mitigating potential risks associated with code generation. I believe that a good programmer with this process could achieve better results than by working by themselves as I do believe CoPilot can be a very useful tool, it just can not be trusted blindly to provide the answer, the knowledge of how the code works must still be there to fix or re-prompt CoPilot.

6.4 What I learned from doing this project

Firstly, one of the key skills I have developed from this experience is the art of critical reading. Before this, traditional educational projects rarely required such a degree of depth in how to extract, relate and formulate ideas from academic articles or books. Without a doubt, this process of reading and critical analysis has greatly improved my ability to determine, synthesize information and present it in a coherent, consistent and attractive way. This critical analysis helped me tell and determine what I liked and did not like about the articles, giving me a rich perspective on how to write my own.

On the other hand, the planning of scenarios and test methods was challenging, the development of my investigative capacity that challenged me to critically analyze and confront the articles, seeking objective evidence of the results and developing recommendations, allowed me to explore facts and phenomena, analyze problems, observe, collect and organize relevant information, to collect and analyze data and report results. During the process of creating my paper, I worked on the overall planning and organization of my article and its different sections. I greatly appreciated Andrew's input and his insistence on using a 'scaffolding' method that greatly facilitated this process, and which I will capitalize on for future writing tasks.

After seeing the performance and results of CoPilot, I can not deny that my initial enthusiasm for large language models (LLMs) waned. I had anticipated that CoPilot would face difficulties or not meet expectations, but I was not prepared for the magnitude of it's mistakes, which seemed minor but were mistakes I would never have made. It was surprising how difficult it was to identify errors when the code looked correct at first glance. This experience has made me reconsider my expectations for LLMs, highlighting the complexities and potential dangers of relying solely on automated tools for code translation and optimization.

6.5 Future Works

It would be interesting to try this again using Devin https://www.cognition-labs.com/introducing-devin from cognition labs as it has shown to perform better than most LLMs when used for coding. From the suggested performance, there could be a different result in redoing this research with Devin as the translator 6.2.

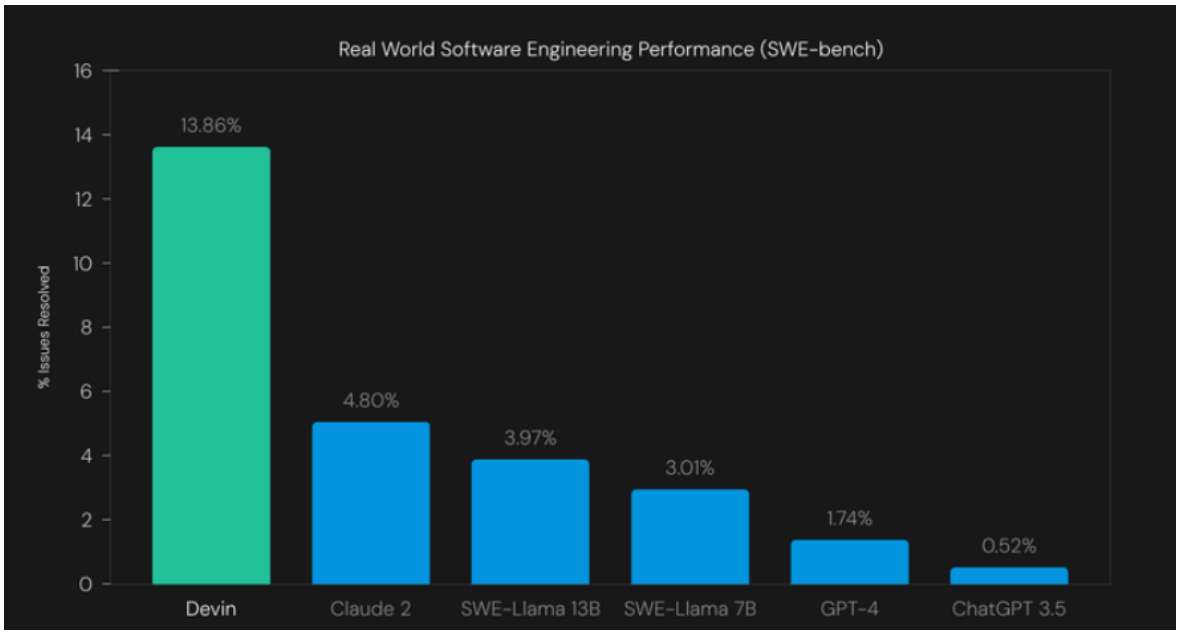


Figure 6.2: Devin's Performance

Another interesting project would be translating ViSQOL from Matlab to Python, and implementing Warp-Q in Matlab. This project could significantly enhance CoPilot's performance in translating speech and audio models. Matlab and Python are both dynamically typed programming languages, which means variables do not require explicit typing. This flexibility in coding allows for various approaches, but it also poses challenges in terms of code readability and maintainability. This could result in very different results than those found in my paper, but I am not sure.

Acknowledgements

Thanks to Andrew Hines for helping me do this project.

Thanks to Fabian Carrocera for proof reading.

Thanks to Telma Munoz for her help with the image, graph selection and proof reading.

Code repository

Bibliography

1. Statista. *Worldwide Developer Survey: Most Used Languages* Accessed on 29 November 2023. https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/.
2. Abelson, H., Sussman, G. J. & Sussman, J. *Structure and Interpretation of Computer Programs* 2nd. ISBN: 0-262-01077-1 (MIT Press, Cambridge, MA, USA, 1984).
3. Wikipedia contributors. *History of Programming Languages* Accessed on 29 November 2023. https://en.wikipedia.org/wiki/History_of_programming_languages.
4. *Computer Languages* 3rd December, 2023. https://www.cs.mtsu.edu/~xyang/2170/computerLanguages.html.
5. Knuth, D. E. *The Art of Computer Programming* Volumes 1-3. ISBN: 0-201-89683-4 (Addison-Wesley, Reading, MA, USA, 1968).
6. *Computer Languages* 3rd December, 2023. https://www.zdnet.com/article/top-programming-languages-most-popular-and-fastest-growing-choices-for-developers/.
7. Martin, R. C. *Clean Code: A Handbook of Agile Software Craftsmanship* ISBN: 978-0-13-235088-4 (Prentice Hall, Upper Saddle River, NJ, USA, 2008).
8. Gabbrielli, M. *Programming Languages: Principles and Paradigms* (2005).
9. Scott, M. L. *Programming Language Pragmatics* (2015).
10. Alomari, Z., El Halimi, O., Sivaprasad, K. & Pandit, C. Comparative Studies of Six Programming Languages.
11. Wikipedia contributors. *Comparison of programming languages* Accessed on 29 November 2023. https://en.wikipedia.org/wiki/Comparison_of_programming_languages.
12. Vogel, P. *Why Commenting Code Is Still Bad* https://visualstudiomagazine.com/articles/2013/07/26/why-commenting-code-is-still-bad.aspx.
13. Spertus, E. *Best Practices for Writing Code Comments* https://stackoverflow.blog/2021/12/23/best-practices-for-writing-code-comments/.
14. Wikipedia contributors. *Large Language Model* Accessed on 29 November 2023. https://en.wikipedia.org/wiki/Large_language_model.
15. Peng, B. *et al.* Check your facts and try again: Improving large language models with external knowledge and automated feedback. *arXiv preprint arXiv:2302.12813* (2023).
16. Google. *Introduction to Large Language Models* Accessed on 29 November 2023. https://developers.google.com/machine-learning/resources/intro-llms.
17. Ogundare, O. & Araya, G. Q. Comparative Analysis of CHATGPT and the Evolution of Language Models. *Journal of Natural Language Processing* (2023).
18. Naveed, H. *et al.* A Comprehensive Overview of Large Language Models. *Journal of Artificial Intelligence Research* (2023).
19. Omiye, J. A., Gui, H., Rezaei, S. J., Zou, J. & Daneshjou, R. Large language models in medicine: the potentials and pitfalls. *arXiv preprint arXiv:2309.00087* (2023).
20. SEI Insights. *Application of Large Language Models (LLMs) in Software Engineering: Overblown Hype or Disruptive Change?* Accessed on 29 November 2023. https://insights.sei.cmu.edu/blog/application-of-large-language-models-llms-in-software-engineering-overblown-hype-or-disruptive-change/.

21. GitHub. *Universe 2023: Copilot transforms GitHub into the AI-powered developer platform* Accessed on 29 November 2023. https://github.blog/2023-11-08-universe-2023-copilot-transforms-github-into-the-ai-powered-developer-platform/.

22. GitHub. *GitHub Copilot Documentation* Accessed on 29 November 2023. https://docs.github.com/en/copilot/overview-of-github-copilot/about-github-copilot-individual.

23. Hines, A., Skoglund, J., Kokaram, A. C. & Harte, N. ViSQOL: an objective speech quality model. *EURASIP Journal on Audio, Speech, and Music Processing* **2015,** 1–18 (2015).

24. QxLabIreland. *WARP-Q: Wireless Open-Access Research Platform for Quantum Communications* https://github.com/QxLabIreland/WARP-Q.

25. Wikipedia contributors. *Audio signal processing* Accessed on 29 November 2023. https://en.wikipedia.org/wiki/Audio_signal_processing.

26. Reynolds, D. A. *et al.* Gaussian mixture models. *Encyclopedia of biometrics* **741** (2009).

27. De Souza, C. R., Redmiles, D., Cheng, L.-T., Millen, D. & Patterson, J. *Sometimes you need to see through walls: a field study of application programming interfaces* in *Proceedings of the 2004 ACM conference on Computer supported cooperative work* (2004).

28. Santoro, M. *et al. Web Application Programming Interfaces (APIs): General-Purpose Standards, Terms and European Commission Initiatives* tech. rep. (Technical Report. European Commission, Louxembourg, 2019).

29. Piccioni, M., Furia, C. A. & Meyer, B. *An empirical study of API usability* in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (2013), 5–14.

30. GeeksforGeeks. *Merge Sort* https://www.geeksforgeeks.org/merge-sort/?ref=header_search (2024).

31. GeeksforGeeks. *Quick Sort* https://www.geeksforgeeks.org/quick-sort/?ref=shm (2024).

32. GeeksforGeeks. *Python Program for Dijkstra's Shortest Path Algorithm | Greedy Algo-7* https://www.geeksforgeeks.org/shortest-distance-between-two-nodes-in-graph-by-reducing-weight-of-an-edge-by-half/?ref=header_search (2024).

33. NumPy Contributors. *NumPy Documentation* Accessed: 26th March 2024. https://numpy.org/doc/stable/.

34. Sanderson, C. *Armadillo C++ Library Documentation* Accessed: 26th March 2024. http://arma.sourceforge.net/docs.html.

35. Wouters, D. *libfvad: Free Voice Activity Detector C++ Library* https://github.com/dpirch/libfvad.

36. Siciarz, Z. *Aquila: A Digital Signal Processing Library for C++* https://github.com/zsiciarz/aquila. Accessed: 2024.

37. *Essentia* https://essentia.upf.edu/. Accessed: April 2, 2024.

38. *Kaldi ASR* https://kaldi-asr.org/. Accessed: April 2, 2024.